D Verteilte Objekte und CORBA

D.1 Überblick

- Verteilte Systeme
- OOP und Verteilung
- Java RMI
- CORBA
  - ➤ Architekturüberblick
  - ➤ Der Object Request Broker (ORB)
    (Interface Description Language (IDL), Remote invocation, Dynamic Invocation, Komponenten des ORB)
  - ➤ Portable Object Adaptor
  - ➤ CORBA Services

D.2 Verteilte Systeme

- *"Distributed System"*
  Definition nach Tanenbaum und van Renesse
  - ◆ It looks like an ordinary centralized system.
  - ◆ It runs on multiple, independent CPUs.
  - ◆ The use of multiple processors should be invisible (transparent).
- *"Distributed System"*
  Definition nach Mullender
  - ◆ Zusätzlich: Not any single points of failures
- Definitionen sind nicht präzise
  - ◆ Manchmal ist es schwierig, ein lokales bzw. verteiltes System zu identifizieren
  - ◆ Definitionen basieren oft auf bestimmten Eigenschaften, die gerade wichtig erscheinen

Literatur

NeS98. J. Nehmer, P. Sturm: *Systemsoftware, Grundlagen moderner Betriebssysteme*. dpunkt, 1998.

Mul89. S. Mullender (Ed.): *Distributed Systems*. ACM Press, 1989.

Tan94. A. S. Tanenbaum, M. van Steen: *Distributed Systems*. Prentice Hall, 2002.

Tan95. A. S. Tanenbaum: *Verteilte Betriebssysteme*. Prentice Hall, 1995.

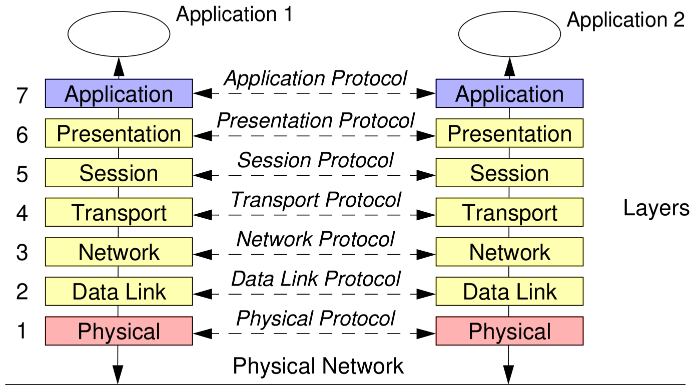
BiN84. A. D. Birrel, B. J. Nelson: "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems* **2**(1), Feb. 1984, pp. 39–59.

Flyn72. M. J. Flynn: "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*, **C-21**, Sept. 1992, pp. 948–960.

D.3 Kommunikationsmodelle

- Kommunikation erfordert gemeinsame Sprache
  - ◆ Protokolle

1 Protokollschichten nach dem ISO OSI Referenzmodell



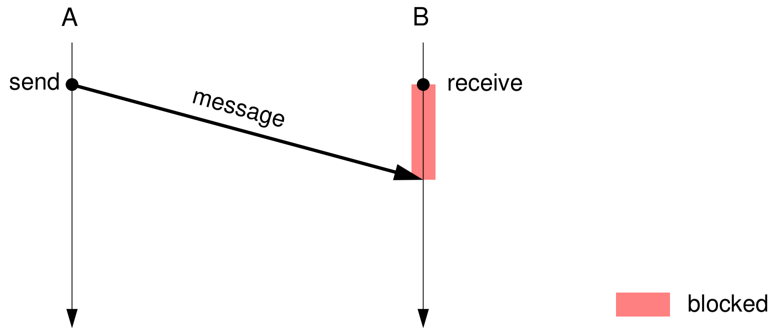
- Physical Layer (Physikalische Schicht)
  - Übertragung von 0-en und 1-en über ein Kabel
- Data Link Layer (Verbindungsschicht)
  - Senden von Bits; Trennung von Rahmen oder Paketen; Fehlerprüfung
- Network Layer (Netzwerkschicht)
  - Nachrichten durch größere Netze leiten (Routing)
- Transport Layer (Transportschicht)
  - Implementation von zuverlässigen Verbindungen
  - Fragementierung und Zusammensetzen von Paketen
- Session Layer (Sitzungsschicht)
  - Dialogkontrolle, Synchronisation
- Presentation Layer (Präsentationsschicht)
  - Transparenz von unterschiedlicher interner Datendarstellung
- Application Layer (Anwendungsschicht)
  - Menge von Anwendungsprotokollen
  - Electronic mail protocol
  - File transfer protocol
  - HTTP
  - etc.

2 Klassifikation

- Synchronität
  - ◆ wird der Sender blockiert bis der Empfänger die Nachricht hat, oder nicht?
- Interaktionsmuster
  - ◆ Message Passing — eine Nachricht wird von einem Teilnehmer an einen anderen gesendet
  - ◆ Request-Reply (Client-Server) Interaktion — Nachricht an einen Empfänger und Nachricht zurück an den ursprünglichen Absender
- Adressaten
  - ◆ Ein Empfänger
  - ◆ Mehrere Empfänger (Gruppenkommunikation, Multicast, Broadcast)

2 Datagramm-Nachricht

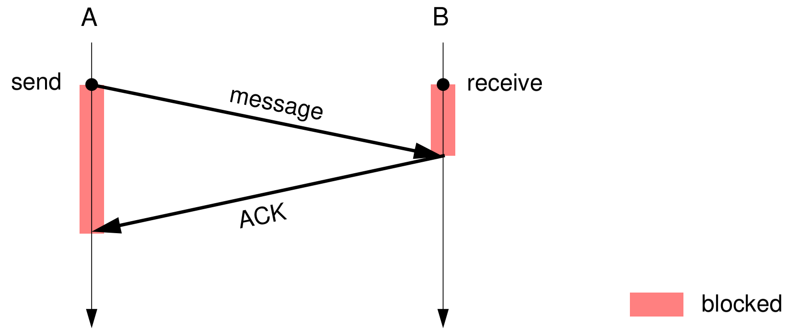
- Message-Passing; asynchrones Senden



- ◆ Sender kann sofort weiterarbeiten
- ◆ Empfänger ist blockiert bis eine Nachricht eintrifft
- ◆ Benötigt Pufferspeicher für noch nicht angenommene Nachrichten

3 Rendezvous-Modell

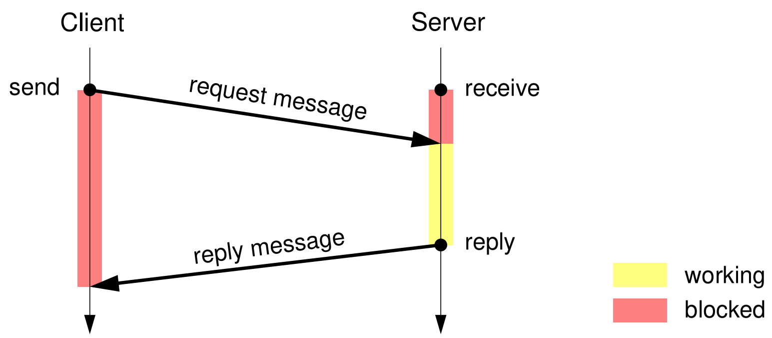
- Message-Passing; synchrones Senden



- ◆ Sender wartet bis Nachricht empfangen wurde
- ◆ Empfänger ist blockiert bis eine Nachricht eintrifft
- ◆ benötigt keinen Pufferspeicher

4 Synchrones Request-Reply Modell

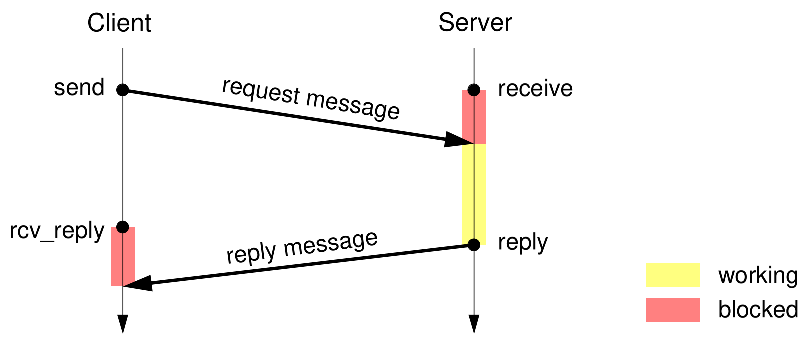
- Request-Reply Interaktion; synchrones Senden



- ◆ Sender wartet bis Antwort-Nachricht empfangen wurde
- ◆ Empfänger ist blockiert bis eine Nachricht eintrifft
- ◆ Client und Server arbeiten nicht nebenläufig
- ◆ bekannteste Realisierung ist RPC (Remote Procedure Call)

5 Asynchrones Request-Reply Modell

- Request-Reply Interaktion; asynchrones Senden



- ◆ Client und Server arbeiten nebenläufig
- ◆ Basis für Gruppenkommunikation

6 Zuverlässigkeit

- Nachrichten können verloren gehen wenn keine zuverlässige Verbindung benutzt wird
  - ◆ zuverlässige Verbindungen benutzen Acknowledge-Nachrichten (ACK)
  - ◆ für einfache Nachrichtenübertragung großer Overhead

★ Zuverlässigkeit mit Request-Reply Interaktionsmodell kombinieren

- Mögliche Fehler
  - ◆ Server Crash
    Fehlermodell: Totalamnesie
    (Server verliert alle Informationen über frühere Anfragen)
  - ◆ Anfrage-Nachricht geht verloren
  - ◆ Antwort-Nachricht geht verloren
- Ideale Semantik
  - ◆ *exactly-once*
    die Anfrage wird genau einmal auf Serverseite bearbeitet

6 Zuverlässigkeit (2)

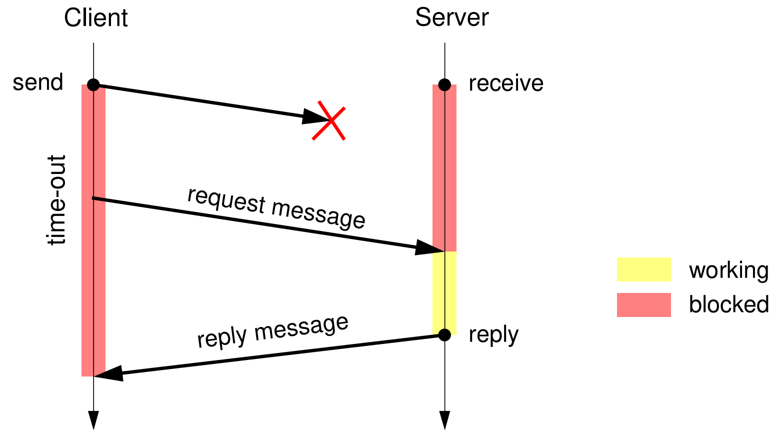
- **At-Least-Once Semantik**
  - ◆ Anfrage wird einmal oder mehrmals bearbeitet
  - ◆ Client bekommt nie eine Fehlermeldung aber er erkennt eventuell, dass die Anfrage mehrfach bearbeitet wurde: Operationen müssen *idempotent* sein!
- Implementation
  - ◆ wenn der Client nach einiger Zeit (time-out) keine Antwort erhält, wiederholt er die Anfrage
    - keine zusätzliche Funktionalität auf Server-Seite erforderlich
    - der Server darf wiederholte Anfrage aber ignorieren, wenn er sie erkennen kann

6 Zuverlässigkeit (3)

- **At-Most-Once Semantik**
  - Die Anfrage wird höchstens einmal (oder überhaupt nicht) bearbeitet
- Einfache Implementation (nur auf der Client-Seite)
  - wenn das Ergebnis innerhalb einer bestimmten Zeit nicht eintrifft wird dem Aufrufer eine Fehlermeldung zurückgegeben (at-most-once Semantik)
  - sonst wird das Ergebnis zurückgegeben (exactly-once Semantik)
- Komplexere Implementation
  - Client wiederholt Nachricht nach einem Time-out (verdeckt Nachrichtenverlust auf der Verbindung)
  - Client muss Server-Abstürze erkennen (Fehlermeldung an den Aufrufer, at-most-once Semantik)
  - Server hebt Antworten auf (Wiederholung bei Verlust der Nachricht)
  - Server muss alte Anfragen nach einem Absturz erkennen und ignorieren
  - wenn das Ergebnis zurückgegeben wird, haben wir exactly-once Semantik

6 Zuverlässigkeit (4)

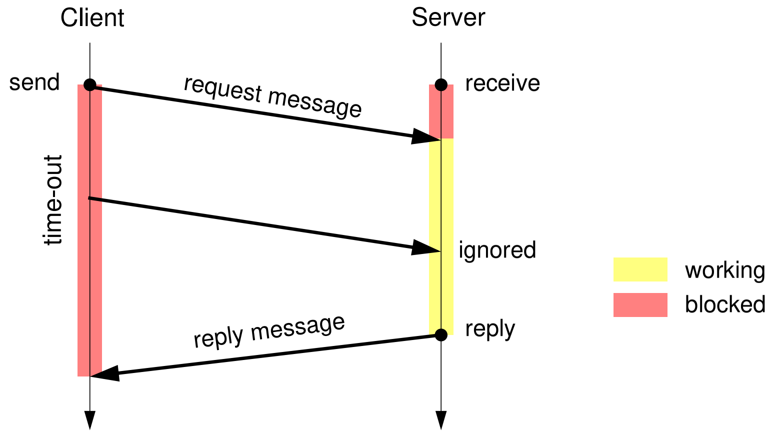
- Anfrage geht verloren



- Anfrage wird wiederholt

6 Zuverlässigkeit (5)

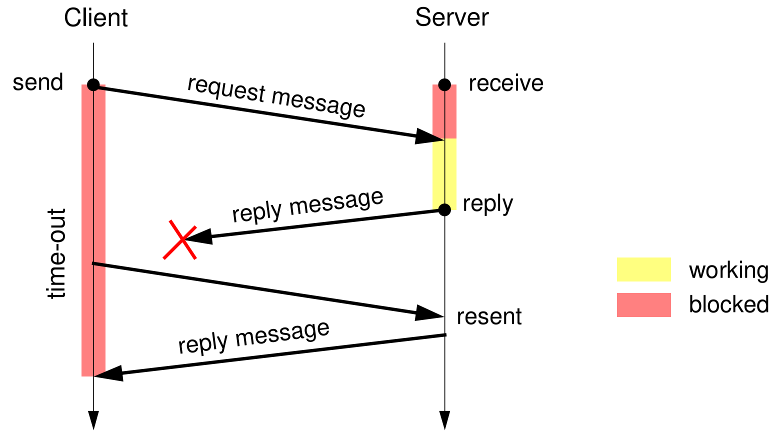
- Bearbeitung ist noch nicht fertig



- wiederholte Anfrage wird ignoriert

6 Zuverlässigkeit (6)

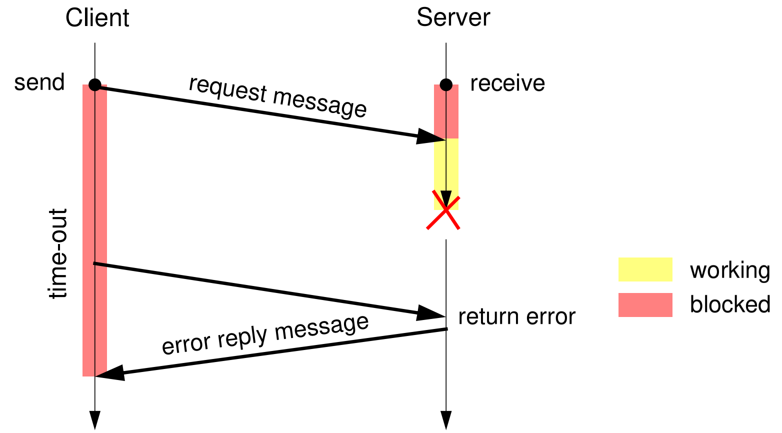
- Antwort-Nachricht geht verloren



- Server hebt Antwort auf und wiederholt sie

6 Zuverlässigkeit (7)

▲ Server stürzt ab



- Server erkennt alte Anfragen (alte Generationsnummer) und liefert Fehler zurück (at-most-once Semantik)

7 Remote Procedure Calls

■ Request-Reply-Modell kann für die Implementierung von RPCs genutzt werden
[Birrell and Nelson 1984]

- statt eine Anfrage zu senden wird eine Remote-Prozedur aufgerufen
- statt einer Antwort erhält man die Ergebnisse des Aufrufs

★ Aufruf einer Prozedur ist ortstransparent

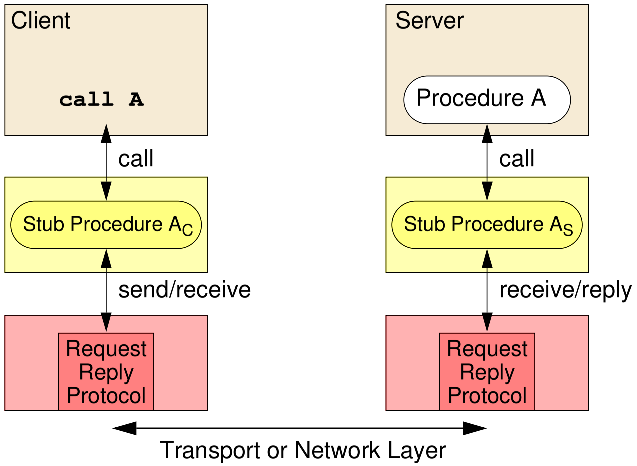
- Syntax für lokale und remote Aufrufe kann identisch sein
- sehr intuitiv
- keine expliziten send- und receive-Anweisungen erforderlich

■ Implementierung von RPCs

- Stub-Prozeduren auf Client- und Server-Seite

7 Remote Procedure Calls (2)

- Implementierung von RPCs mit Stub-Prozeduren



nach Nehmer 1995

7 Remote Procedure Calls (3)

- Client-Stub
  - Marshalling der Parameter (Zusammenstellen einer Anfrage-Nachricht)
  - Senden der Anfrage-Nachricht
  - Warten auf die Antwort-Nachricht
  - Unmarshalling des Ergebnisses
  - Implementierung der Zustell-Semantik
- Server-Stub
  - Empfangen der Anfrage-Nachricht
  - Unmarshalling der Parameter
  - Aufruf der Server-Prozedur
  - Marshalling der Ergebnisse
  - Senden der Antwort-Nachricht
  - Implementierung der Zustell-Semantik

7 Remote Procedure Calls (4)

▲ Probleme bei RPCs
- Marshalling der Parameter
  - Zahl und Typen müssen bekannt sein (vgl. mit C: `printf( "Count %d\n", count )`)
- Parameter-Übergabe-Semantik
  - *Call-by-value:* unproblematisch
  - *Call-by-reference:* wie implementieren?
- keine globalen Variablen
- Semantik
  - Server-Absturz; keine exactly-once Semantik
- Performance
  - keine Nebenkläufigkeit
  - große Parameter-Daten
  - kurze Prozeduren

7 Remote Procedure Calls (5)

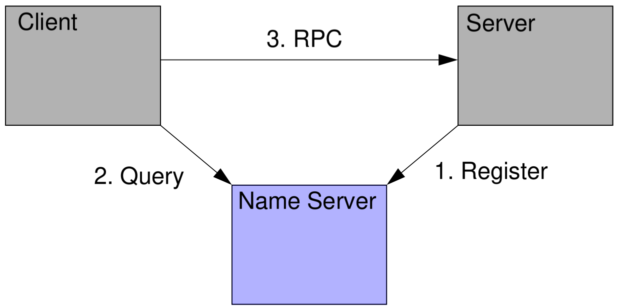
■ Automatische Erzeugung von Stub-Prozeduren
- Werkzeug kann Code generieren für:
  - Parameter-Marshalling
  - Client-Stub Prozedur
  - Server-Stub Prozedur
  - Server-Schleife zum Warten auf Anfragen

■ Binden von Client-Stubs an Server-Stubs
- Server-Stub hat Netzwerk-Adresse, die Client-Stub kennen muss
- Problem: woher kennt der Client den Server?

★ Name-Server
- Symbolische Namen werden in Netzwerk-Adressen umgesetzt

8 Name-Server und Binden

- Bekannter Name-Server wandelt Namen in Adressen um
  - ◆ Client kennt Namen seines Servers und die Adresse eines Name-Servers
  - ◆ Name-Server wandelt Namen in eine (dynamische) Netzwerk-Adresse um
  - ◆ Client kann sich immer an Server binden
  - ◆ Server muss seine Netzwerkadresse beim Name-Server registrieren



D.4 OO Verteilte Anwendungen

- Objektorientierte Anwendungen im verteilten System
- Verteilung auf
  - ➤ verschiedene Rechner
  - ➤ verschiedene Prozessoren
  - ➤ verschiedene virtuelle Maschinen
  - ➤ verschiedene Adressräume
    - Auch wenn nur die ersten beiden Punkte die Definitionen von Verteilten Systemen treffen, treten auch bei einer Verteilung auf verschiedene virtuelle Maschinen oder verschiedene virtuelle Adressräume auf einer CPU ähnliche Probleme auf: Es gibt eine Grenze zwischen Objekten, die von der "normalen Objektinteraktion" (=Methodenaufruf) nicht ohne besondere Vorkehrungen überwunden werden kann.
- Objektinteraktionen zwischen objektorientierten Programmen *oder* Aufteilung eines Programmes in interagierende Programmteile?
  - In beiden Fällen interagieren Objekte über Grenzen hinweg. Meist ist eine solche Interaktion nicht von der jeweiligen Programmiersprache direkt unterstützt, so dass zusätzliche Mechanismen ins Spiel kommen müssen.

D.5 OOP und Verteilung

1 Klassifikation von Interaktionsformen

- Formen/Ausprägungen von Objektinteraktion im Verteilten System
  - ◆ explizit
    - der Anwendungscode beschreibt die Interaktion im verteilten System explizit (z.B. durch Aufbau einer Verbindung zu einem anderen Rechner)
  - ◆ implizit
    - die Interaktion im verteilten System erfolgt bei Bedarf implizit (z. B. wenn eine Objektreferenz eine Remote-Referenz ist)
  - ◆ orthogonal
    - die Programmiersprache enthält keine Sprachkonzepte zur Unterstützung verteilter Anwendungen. Die Interaktion zwischen verteilten Objekten erfolgt mit Hilfe von Klassen- oder Funktionsbibliotheken, die Kommunikationsmechanismen bereitstellen
  - ◆ nicht-orthogonal
    - die Programmiersprache enthält Sprachkonstrukte, die eine Interaktion zwischen Objekten im verteilten System erlauben
  - ◆ uniform
    - lokale und verteilte Objektinteraktion unterscheiden sich auf programmiersprachlicher Ebene nicht
  - ◆ nicht-uniform
    - die Programmiersprache unterscheidet lokale und verteilte Objektinteraktion
  - ◆ transparent
    - die Verteilung ist für den Anwendung transparent, bei der Programmierung muss in Bezug auf Verteilung nichts berücksichtigt werden
  - ◆ nicht-transparent
    - die Verteilung ist nicht transparent, selbst bei uniformer und impliziter Objektinteraktion im verteilten System muss der Programmierer Vorkehrungen getroffen haben (z.B. Abfangen von Remote-Exceptions)

2 explizite, orthogonale Interaktion

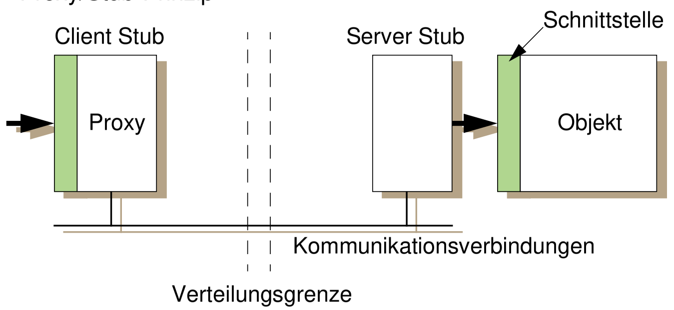
- weit verbreitete Vorgehensweise
- von klassischen Interprozesskommunikations-Mechanismen geprägt
  - ➤ Nachrichten (Datagramm-Sockets, Messages, …)
  - ➤ Verbindungen (Stream-Sockets, Pipes, …)

\+ Vorteile
- ◆ meist weit verbreitete, etablierte Infrastruktur vorhanden
- ◆ kein "unsichtbarer" Overhead

– Nachteile
- ◆ Programmierung aufwendig
- ◆ Bruch im objektorientierten Paradigma
  - ➤ Serialisierung von Parametern, Verlust von Typ-Information
- ◆ Verteilung wird durch die Programmierung "fest verdrahtet"
  - ➤ Software sehr unflexibel in Bezug auf Änderungen

3 implizite, nicht-orthogonale Interaktion

- Interaktion zwischen lokalen und verteilten Objekten unterscheidet sich prinzipiell nicht
  - ➤ nur ein Interaktionsmechanismus: Methodenaufruf
- grundlegendes Konzept: Remote Procedure Call
  - ➤ Verteilung wird durch Vermittler- oder Stellvertreterobjekte vor den Kommunikationspartnern (weitgehend) verborgen
  - ➤ Proxy/Stub-Prinzip



- Ein Stub ist ein Vermittler für Methodenaufrufe, ein Proxy ist ein Stellvertreter für ein Objekt (die Begriffe werden häufig synonym gebraucht, ganz genau genommen nehmen die Stubs die Funktion von Proxies wahr).
- Mit Hilfe von Stubs kann die Objektinteraktion über Adreßraumgrenzen realisiert werden. Auf der Client-Seite gibt es das Proxy- oder Client-Stub-Objekt. Es stellt die gleiche Schnittstelle wie das anzusprechende Objekt selbst bereit. Über einen Kommunikationskanal, der z.B. durch UNIX-Sockets oder ähnliches realisiert wird, kommuniziert der Client-Stub mit dem Server-Stub, der letztlich den am Proxy registrierten Methodenaufruf am richtigen Objekt durchführt.
- Der Client-Stub ist ein Stellvertreter für das eigentliche Objekt, der Server-Stub ein Stellvertreter für den Aufrufer. Der Server-Stub wird häufig auch als "Skeleton" bezeichnet.

4 uniforme / nicht-uniforme Interaktion

★ **<u>nicht-uniforme Interaktion</u>**

- unterschiedliche Methodenaufrufe für lokale und remote-Referenzen
- unterschiedliche Semantik bei der Parameterübergabe
  - ➤ by reference, by value
  - ➤ Problem: Übergabe von lokalen Objektreferenzen

★ **<u>uniforme Interaktion</u>**

- keinerlei Unterschied zwischen lokalen und remote-Aufrufen

5 transparente / nicht-transparente Verteilung

- volle Transparenz:
  Anwendungsentwickler sieht keinerlei Unterschied zwischen lokalen und verteilten Objekten
- Probleme:
  - im verteilten Fall können spezielle Fehler auftreten
    - ➤ unabhängiger Objektausfall →Remote Exception
  - verteilte Interaktion ist implizit signifikant teurer
    - ➤ Transparenz kann zu Ineffizienz führen
  - Verteilung ist häufig ein Entwurfskriterium
    - ➤ Verbergen der Verteilung in der Implementierung ist unsinnig
- Fazit:
  - bei der Programmierung sollte zwischen potentiell verteilten und definitiv lokalen Objekten unterschieden werden können

6 Herausforderungen

- Überwindung heterogener Hardware- und Softwarestrukturen
  - ➤ verschiedene Hardware
  - ➤ verschiedene Betriebssysteme
  - ➤ verschiedene Programmiersprachen
    - Durch die Verteilung von Programmteilen auf mehrere Rechner entsteht meist automatisch das Problem der Heterogenität der verwendeten Komponenten.
- Ortstransparenz
  - ➤ statische Konfiguration
  - ➤ Objektmigration
    - Ist es unwichtig, auf welchem Rechner ein Objekt nun wirklich implementiert ist, so lassen sich zum einen Konfigurationsänderungen bei der Verteilung ohne das Ändern der Applikation durchführen (statische Änderungen). Zum anderen ist es eventuell auch möglich Objekte zur Laufzeit von einem Rechner zum anderen wandern zu lassen (dynamische Änderung).
- Globale Dienste
  - z.B. Namensdienste, Transaktionsdienst, Persistenz, Kontrolle der Nebenläufigkeit
    - Objekte müssen sich nun finden. Dazu wird ein übergreifender Namensdienst benötigt.
    - Andere höherwertige Dienste können für die verteilte Applikation nicht lokal erbracht werden, sondern müssen übergreifend von allen Rechnersystemen bereitgestellt werden.

D.6 Java RMI

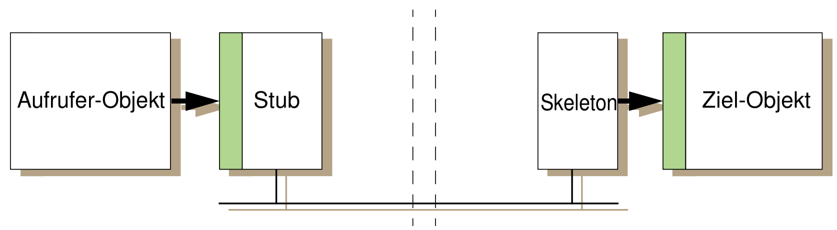
- Infrastruktur zur Unterstützung von verteilten Java-Anwendungen
  - ◆ Server-Anwendung
    - ➤ erzeugt Objekte
    - ➤ macht Objektreferenzen verfügbar
    - ➤ wartet auf Methodenaufrufe
  - ◆ Client-Anwendung
    - ➤ besorgt sich Remote-Referenz
    - ➤ ruft Methoden an entferntem Objekt auf
- Probleme
  - ➤ Entfernte Objekte finden
  - ➤ Methodenauf uf
  - ➤ Übergabe von Objekten

1 Entfernte Objekte finden

- Nameserver zur Umwandlung von Namen in Remote-Referenzen (rmiregistry)
  - ➤ Name = URL, bestehend aus Registry-Host[:Port] und Objektnamen
- Spezielle Klasse **`java.rmi.Naming`** für transparenten Zugriff auf Nameserver
  - ◆ Server meldet Objekt bei seiner Registry an
    - ➤ **`void Naming.bind(String name, Remote obj)`**
      - – registriert ein Objekt unter einem eindeutigen Namen, falls das Objekt bereits registriert ist wird eine Exception ausgelöst
    - ➤ **`void Naming.rebind(String name, Remote obj)`**
      - – registriert ein Objekt unter einem eindeutigen Namen, falls das Objekt bereits registriert ist wird der alte Eintrag gelöscht
  - ◆ Client bekommt Referenz von Registry
    - ➤ **`Remote Naming.lookup(String name)`**
      - – liefert die Objektreferenz zu einem gegebenen Namen
- Alternative: ein Objekt erhält Remote-Referenz als Parameter oder als Ergebnis eines Methodenaufrufs

2 Methodenaufruf

- Klassische Stub-/Skeleton-Technik



- Stub und Skeleton werden aus der Implementierung des Zielobjekts generiert (**`rmic`**)
- Stub-Klasse wird bei Bedarf automatisch vom Server geladen (**`RMIClassLoader`**)
- Ausführungssemantik: "at most once"
  - ➤ bei Fehler wird **`RemoteException`** ausgelöst

3 Parameterübergabe

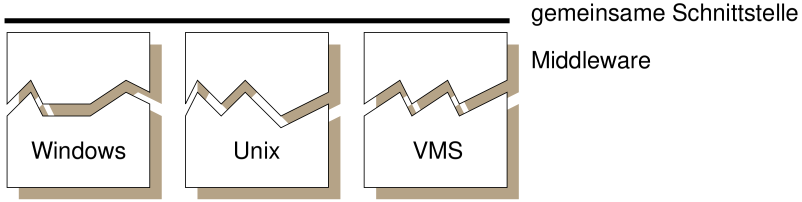
- keine einheitliche Parameterübergabesemantik
  - ◆ Problem: Übergabe von Referenzen auf Objekte, die kein Remote-Interface implementieren
    - ➤ keine Stub- und Skeleton-Klassen vorhanden
  - ◆ Ausweg: unterschiedliche Übergabesemantik
    - ➤ by reference: für alle Objekte von Klassen, die ein Remote-Interface implementieren
    - ➤ by value: für alle anderen Objekte
      - Objektzustand muss allerdings zumindest serialisiert werden können, Klasse kann über RMIClassLoader geholt werden.

4 Resumee

- einfacher, speziell auf Java abgestimmter Fernaufrufmechanismus
- implizite, nicht-orthogonale Interaktion (durch Remote-Referenzen)
- Verteilung nicht transparent

D.7 Middleware für verteiltes Programmieren

- Middleware – Software zwischen Betriebssystem und Anwendung



◆ Bereitstellung von Diensten für verteilte Anwendungen

◆ **gesucht:** Middleware für verteiltes objektorientiertes Programmieren

- Middleware ist klassisch eine Softwarekomponente, die Dienste für verteilte Anwendungen bereitstellt. Dabei abstrahiert Middleware von der eingesetzten Hardware- und Softwarearchitektur und bietet eine gemeinsame Schnittstelle an.

- CORBA – Common Object Request Broker Architecture
  Standard der OMG
  - Gemeinsame Architektur für einen Object Request Broker (ORB). Ein ORB ist eine Vermittlungseinheit für den Aufruf von verteilten Objekten. Das „gemeinsam“ bezieht sich auf die gemeinsame Anstrengung der wichtigsten IT Firmen in diesem Bereich, zusammengeschlossen in der OMG (Object Management Group), einen Standard für verteiltes Programmieren mit Objekten zu entwickeln.

D.8 CORBA — Architektur

1 Überblick

- Motivation
- Object Management Architecture (OMA)
- Anwendungsobjekte und IDL
- Object Request Broker (ORB)
- Portable Object Adaptor (POA)
- CORBA Services

2 Literatur, URLs

OMG99. Object Management Group, OMG: *The Common Object Request Broker: Architecture and Specification*. Rev. 2.3.1, OMG Doc. formal/99-10-07, Oct. 1999.

Pope98. A. Pope: *The CORBA Reference Guide*. Addison-Wesley. 1998.

Linn98. C. Linnhoff-Popien: *CORBA, Kommunikation und Management*. Springer, 1998.

TaBu01. Zahir Taki, Omran Bukhres: *Fundametals of Distributed Object Systems*. Wiley, 2001.

Daneben vor allem online-Informationen

- ➤ OMG
  `http://www.omg.org/gettingstarted/`
  offizielle Seiten der Object Management Group
- ➤ Douglas C. Schmidt
  `http://www.cs.wustl.edu/~schmidt/corba.html`
  sehr gute Informationssammlung

3 Motivation

- verschiedene Orte im verteilten System
  - ➤ Ortstransparenz
- Heterogenität im verteilten System
  - – unterschiedliche Hardware
  - – unterschiedliche Betriebssysteme und
  - – unterschiedliche Programmiersprachen
  - ➤ Transparenz der Heterogenität
- Dienste im verteilten System
  - ➤ Namensdienst
  - ➤ Zeitdienst
  - ➤ Transaktionsdienst
  - ➤ …

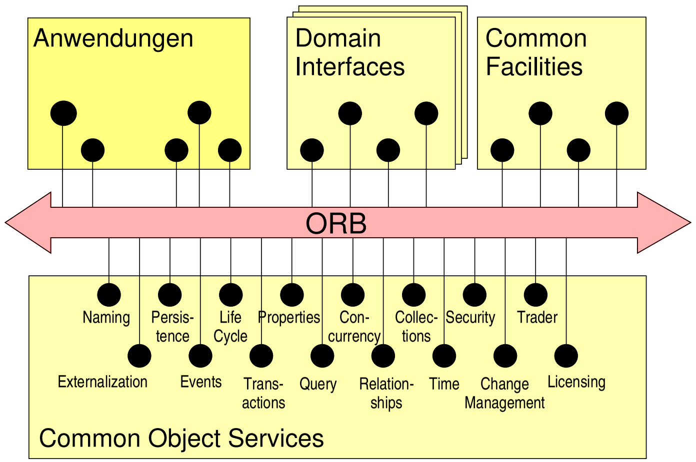
4 Entwurfsziele

- CORBA ist ein Standard
  - Standard-Dokumente
  - Definition von "CORBA compliance"
  - Hersteller realisieren Implementierungen des Standards (z. B., VisiBroker, Orbix, Orbacus, MICO, etc.)
- CORBA ist eine Middleware zur Abstraktion von
  - Hardware,
  - Betriebssystem und
  - Programmiersprache

4 Entwurfsziele (2)

- Interoperabilität
  - Eine Anwendung soll ohne Änderungen auf verschiedenen CORBA-Implementierungen ablaufen können
  - Anwendungen auf verschiedenen CORBA-Implementierungen sollen miteinander kommunizieren können
    - Mit Hilfe von CORBA soll nicht nur von Hardware, Betriebssystem und Programmiersprache abstrahiert werden, sondern auch von der Implementierung des CORBA Systems selbst.
- Einbindung von Altapplikationen ("Legacy-Anwendungen")
  - Kapselung von Altanwendungen in CORBA Objekte
    - Ein CORBA Objekt verdeckt die Altapplikation. Aufrufe an dem Objekt werden in entsprechende Interaktionen mit der Altapplikation umgesetzt.
- Vision der Business Objects / Components
  - alle Geschäftsdaten und -vorfälle sind CORBA Objekte
    - Damit würde die gesamte Geschäftswelt die „gleiche Sprache“ sprechen und mit Hilfe von CORBA-Objektinteraktionen miteinander in Kontakt stehen können.

5 OMA – Object Management Architecture



- Die CORBA Architektur besteht aus dem ORB (Object Request Broker). Dieser vermittelt Aufrufe zwischen Objekten. Der ORB weiß wo welches Objekt liegt. Die Anwendungsobjekte können untereinander mit Hilfe des ORB kommunizieren. Die Anwendungsobjekte stehen untereinander in Client/Server-Beziehung.
- Die Common Object Services sind allgemeine, anwendungsunabhängige, standardisierte Systemdienste, die eine CORBA Implementierung anbieten kann bzw. muss, z. B. den Naming Service zum Auffinden von Objekten anhand eines Namens oder einen Transaktionsdienst zur Unterstützung von Transaktionen im verteilten System. CORBA-Services verhalten sich wie normale Objekte.
- Common Facilities sind ähnlich wie Object Services unabhängig von einem bestimmten Anwendungsbereich. Im Gegensatz zu Services legen sie aber nicht Schnittstellen für Systemdienste, sondern Schnittstellen zu Benutzeranwendungen (z. B. für Dokumentenbearbeitung oder Grafikschnittstellen) fest.
- Domain Interfaces legen ähnlich wie Facilities Anwendungsschnittstellen fest, die sich allerdings an bestimmten Anwendungsbereichen oder Branchen orientieren. Beispielsweise Product Data Management (PDM), Telekommunikation, Dienste des Gesundheitswesens oder Finanzanwendungen.

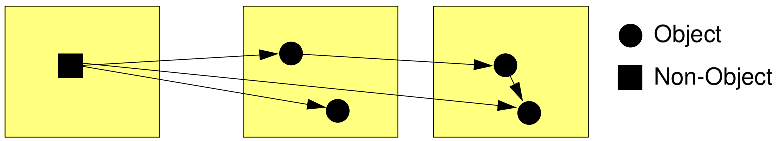
6 CORBA-Implementierungen

- Eine CORBA-Implementierung muss enthalten:
  - die Implementierung der Kern-Architektur
  - eine Sprachabbildung (z. B. für C++)
- Eine CORBA-Implementierung kann ausserdem enthalten:
  - eine beliebige Zahl von Services
  - Funktionalität für die Interoperabilität mit anderen CORBA-Implementierungen (GIOP, IIOP)
- **Wie** die Implementierung die Anforderungen des Standards realisiert bleibt freigestellt
  - unterschiedliche Implementierungen sind möglich
    - als Daemon
    - als Bibliothek
    - ...

D.9 CORBA-Anwendungsobjekte

1 Verteilte Objekte

- ➤ Identität
- ➤ Zustand
- ➤ Methoden
- ➤ CORBA-Objekte können aufgerufen werden (realisieren Server-Seite)
- ➤ CORBA-Objekte können als Client agieren



- ➤ Clients müssen keine Objekte sein (können auch Prozesse, etc. sein)
  - Da Clients keine Objekte sein müssen, muss man kein CORBA-Objekt erzeugen, um mit einem anderen Objekt in Interaktion zu treten.

Achtung:
Server-Objekt darf nicht mit einem CORBA-Server verwechselt werden.

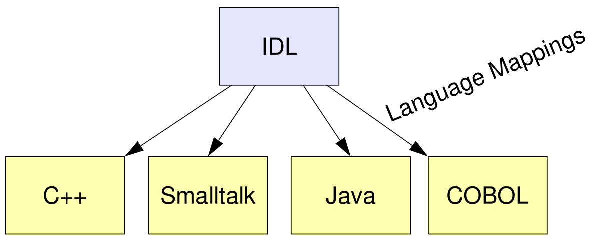
- Server-Objekte sind Server im Sinne der Client/Server-Interaktion. Sie werden über Methodenaufrufe angesprochen.
- CORBA-Server sind Einheiten des darunterliegenden Betriebssystems, die CORBA-Objekte beherbergen, z.B. UNIX-Prozesse.

1 Verteilte Objekte (2)

- ■ Verteilte Objekte bilden eine Anwendung
  - ➤ Anwendungsobjekte liegen auf verschiedenen Rechnern
    - Die Anwendungsobjekte sind die Einheiten der Verteilung in CORBA. Objekte auf einem Rechner können auf eine oder mehrere Systemeinheiten verteilt sein, z. B. auf mehrere Prozesse eines UNIX Systems.
  - ➤ Sie finden sich und kommunizieren miteinander
    - CORBA bietet den Objekten eine transparente Kommunikation, d.h. die Objekte müssen nicht wissen, wo der jeweilige Kommunikationspartner liegt.
- ★ Beispiel Druckmanagement-System
  - ➤ Client-,
  - ➤ Spooler- und
  - ➤ Druckerobjekte
    - Verschiedene Anwendungsobjekte liegen verteilt im System. Sie sind für den Zugriff auf das Drucksystem (Clients), für das Puffern von Druckaufträgen (Spooler) und für das eigentliche Drucken zuständig.
- ■ implizite, nicht-orthogonale Interaktion
  - ◆ (Remote-)Methodenaufrufe
  - ◆ Client-Stub vermittelt Aufruf an Server-Objekt
  - ◆ **At-most-once** / **exactly-once** Semantik

2 Interface Definition Language (IDL)

- Sprache zur Beschreibung von Objekt-Schnittstellen
  - unabhängig von der Implementierungssprache des Objekts
  - Sprachabbildung (Language mapping) definiert, wie IDL-Konstrukte in die Konzepte einer bestimmten Programmiersprache abgebildet werden
  - Language mapping ist Teil des CORBA standards
  - Language mappings sind festgelegt für: C, C++, Smalltalk, COBOL, Ada und Java
  - IDL ist an C++ angelehnt



2 Interface Definition Language (2)

- Beispiel

```
module MyModule
{
    interface MyInterface
    {
        attribute long lines;
        void printLine( in string toPrint );
    };
```

IDL to C++

```
namespace MyModule {
    class MyInterface : ... {
        public:
         virtual CORBA::Long lines();
         virtual void lines( CORBA::Long _val );
         void printLine( const char *toPrint );
        ...
    };
```

2 Interface Definition Language (3)

- Beispiel

```
module MyModule
{
    interface MyInterface
    {
        attribute long lines;
        void printLine( in string toPrint );
    };
```

IDL to Java

```
package MyModule;
public interface MyInterface extends ... {
    public int lines();
    public void lines( int lines );
    public void printLine(java.lang.String toPrint );
    ...
};
```

```
nam
    ...
};
```


2 Interface Definition Language (4)

★ Vorteile

- Unabhängigkeit/Transparenz von der Implementierungssprache
- Ermöglicht Interoperabilität über Sprachgrenzen

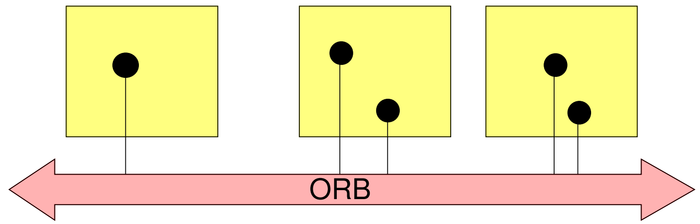
▲ Nachteile

- Objektschnittstellen müssen in der Zielsprache **und** in IDL spezifiziert werden
- IDL ist sehr ausdrucksstark
  - Language mapping für Sprachen, die nicht die entsprechenden Mechanismen anbieten (z. B. C) kann sehr komplex sein
- Wenn eine Sprache spezielle Eigenschaften anbietet, können diese nicht genutzt werden, weil sie von IDL nicht erfasst werden

3 Objekte Erzeugen und Binden

- Erzeugen eines Server-Objekts
  - ◆ Beschreibung der Objektschnittstelle in IDL
  - ◆ Programmierung des Server-Objekts in der Implementierungssprache
  - ◆ Registrierung des Objekts am ORB (bzw. dem Object Adaptor)
    - ➤ der ORB erzeugt eine "Interoperable Object Reference" (IOR)
- Binden des Clients an das Server-Objekt
  - ◆ Referenz auf Server-Objekt besorgen
    - ➤ Ergebnis einer Namensdienst-Anfrage
    - ➤ Rückgabewert eines Methodenaufrufs
    - ➤ von "ausserhalb" des Systsems: Benutzer kennt Referenz als String (IORs können in Strings konvertiert werden und umgekehrt)
  - ◆ Erzeugung des Client-Stub
  - ◆ Methodenaufruf über Client-Stub

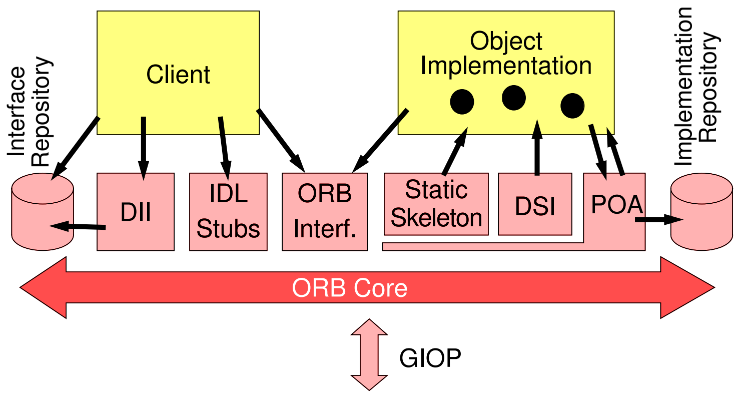
D.10 Object Request Broker – ORB



- The ORB ist das Rückgrat einer CORBA-Implementierung
- Alle Kommunikation zwischen den Objekten läuft über den ORB
  - ◆ ... innerhalb eines Adressraums / Prozesses
  - ◆ ... zwischen Adressräumen / Prozessen
  - ◆ ... zwischen Adressräumen / Prozessen von verschiedenen ORBs
- Der ORB implementiert Ortstransparenz

1 Architektur

- Zentrale Komponenten eines ORB

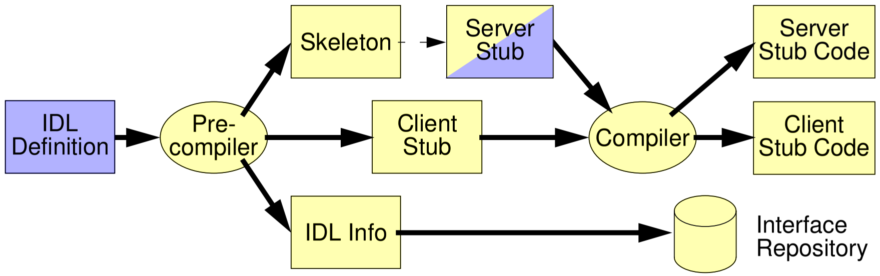


2 Statische Stubs

- Stellvertreter auf der Client- und Serverseite
  - ➤ sobald die Schnittstelle eines Objekts feststeht, können Stubs daraus erzeugt werden
  - ➤ Statische Stubs werden aus der IDL Beschreibung automatisch erzeugt
    - Mittels Precompiler und Compiler werden die Stubs aus der IDL Beschreibung erzeugt. Dabei helfen oft entsprechende Makefiles bei der Erstellung.
  - ➤ Auf Serverseite werden die Stubs (ausgefüllte) *Skeletons* genannt
    - Die Skeletons müssen mit der Implementierung des Objekts (vom Programmierer) ausgefüllt werden. Sie enthalten zunächst lediglich das Methodenskelett.
- Aufgaben der Stubs
  - ➤ Verpacken und Entpacken der Parameter (Marshalling)
  - ➤ Abschicken bzw. Entgegennehmen von Methodenaufruf-Mitteilungen über den ORB-Core

2 Statische Stubs (2)

- Stub-Erzeugung



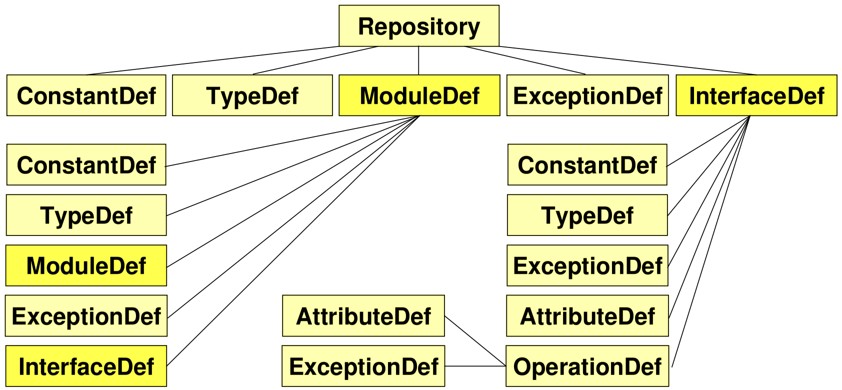
- Prinzipieller Ablauf bei der Definition eines CORBA Objekts
  - Aus der IDL-Definition der Objektschnittstelle werden mittels eines Precompilers mehrere Ausgabedateien erzeugt.
  - Das Skeleton enthält ein Skelett für das Serverobjekt. Es enthält bereits die Server-Stub-Funktionalität und muss um die Implementierung der Methoden des Server-Objekts ergänzt werden. Es bildet dann den Server-Stub inklusive der Implementierung des Objekts selbst.
  - Der Client-Stub enthält den Code für den Client-Anteil.
  - Eine zusätzliche Datei enthält die Informationen, die später in das Interface-Repository geladen werden.
  - Client- und Server-Stub müssen in der Regel noch von einem Compiler übersetzt werden. Die daraus entstehenden Codestücke werden in der Regel dynamisch oder statisch in die Applikation eingebunden.

3 Interface Repository

- Datenbank für Schnittstellendefinitionen in IDL
  - Alle IDL Schnittstellen werden dort abgelegt.
- Abfrage der Datenbank
  - ◆ für Typechecking
    - Der ORB kann prüfen, ob die Schnittstelle des aufgerufenen Objekts mit der Schnittstelle übereinstimmt, die der Client-Stub des Objekts repräsentiert.
    - ➤ bei Inter-ORB-Operationen: Hinterlegung in mehreren Repositories
  - ◆ zum Abruf von Metadaten für Clients und Tools: dynamische Aufrufe, Debugging, Klassenbrowser
    - Beispielsweise für dynamische Aufrufe, d.h. für solche, bei denen der Typ des Objekts zur Compilezeit noch nicht feststand, kann mittels Interface Repository der genaue Typ der Schnittstelle, der Operationen und Parameter zur Laufzeit ermittelt werden.
  - ◆ Implementierung der Methode *get_interface* eines jeden Objekts
    - Jedes Objekt hat eine Methode get_interface mit dem sich ein Verweis auf die dazugehörige Schnittstellenbeschreibung ermitteln läßt. Diese Beschreibung kommt dann aus dem Interface Repository.
- Schreiben der Datenbank
  - ◆ durch IDL Compiler
    - Bei der Generierung von Stubs und Language Binding werden auch Informationen zum Laden des Interface Repositories erzeugt.
  - ◆ durch Schreibmethoden
    - Das Interface Repository besitzt auch Schreibmethoden, mit denen explizit Einträge in der Interface Datenbank vorgenommen werden können.

3 Interface Repository (2)

- Gespeicherte Informationen / Aufbau von IDL Dateien

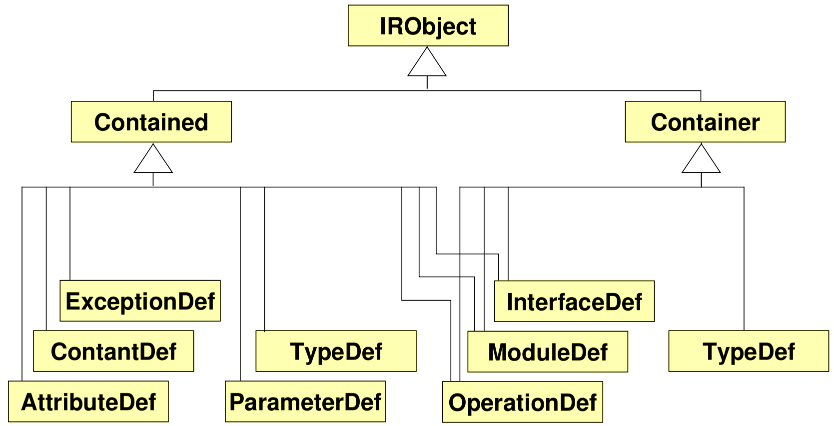


◆ Hierarchie der Komponenten einer IDL Datei bzw. des Interface Repositories

- Die Grafik gibt den strukturellen Aufbau des Repositories wieder. Die Kästchen entsprechen Klassen bzw. Objekten. Die Verbindungslinien sind Assoziationen.
- Ein Module kann beispielsweise Definitionen für Konstanten, Typen, Exceptions, Interfaces und wiederum weiterer Module enthalten.
- Die Struktur entspricht damit dem generellen Aufbau einer IDL-Beschreibung.

3 Interface Repository (3)

- Vererbungshierarchie der IDL Schnittstellen von IR Objekten



- Die Vererbungshierarchie führt zwei neue Klassen ein: Contained und Container. Sie werden dazu benutzt um Objekte darzustellen, die in einem Container enthalten sein können bzw. einen solchen Container darstellen.
- Eine Moduldefinition beispielsweise ist sowohl Container – sie enthält Interfacedefinitionen etc. – als auch Element eines Containers, dem Repository oder eines anderen Moduls.

3 Interface Repository (4)

- Standardtypen werden durch *Type Codes* repräsentiert

TypeCode

  - Repäsentation von Basistypen:
    int, float, boolean
  - Repräsentation von zusammengesetzten Standardtypen:
    union, struct, enum
  - Repräsentation von Template Types und komplexen Deklaratoren:
    sequence, string, array
  - Representation of IDL-based object types (using an interface repository ID)

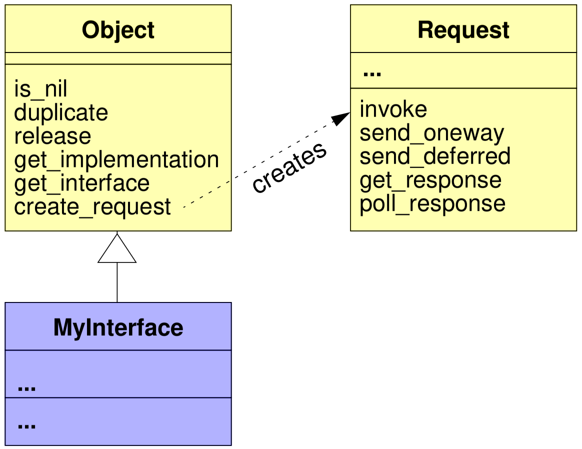
- Typecode repräsentiert den Typ bzw. die Struktur als Objekt mit IDL-Schnittstelle
  - Typecodes werden zur Typprüfung eingesetzt und ergänzen damit die beschreibenden Objekte aus dem Interface Repository. Alle Typen, die sich durch IDL beschreiben lassen werden durch ein entsprechendes Objekt aus dem Interface-Repository selbst repräsentiert, alle Standardtypen durch ein geeignetes Typecode-Objekt.
  - Methode zum Vergleich von Typecodes
  - Methode zur Abfrage einer Typbeschreibung

4 Dynamic Invocation Interface (DII)

- DII ermöglicht Aufruf von Objekten deren Schnittstelle erst dynamisch ermittelt werden können
  - Ermittlung erfolgt über das Interface Repository
- Einzelne Schritte des Aufrufs (im Language Binding abgebildet)
  - ermittle Methodensignatur aus dem Repository
  - erstelle die Parameterliste
  - erstelle die Aufrufbeschreibung (Request)
  - führe Methodenaufruf aus
    - als RPC
      - Es wird der Aufruf generiert und auf das Ergebnis gewartet.
    - asynchroner RPC
      - Es wird der Aufruf generiert. Der Aufrufer kann weiterarbeiten und sich das Ergebnis irgendwann später abholen.
    - über Datagramm-Kommunikation (ohne Antwort)
      - Diese Aufrufart funktioniert nur bei Methoden, die keine Parameter zurückerwarten. Der Aufruf wird generiert; auf ein Ergebnis bzw. den Abschluß der Methodenausführung kann nicht gewartet werden. Hinzu kommt, dass der Aufruf eventuell über einen Datagrammdenst versandt wird. Das bedeutet, daß nicht sichergestellt ist, ob der Aufruf überhaupt durchgeführt wird.

4 Dynamic Invocation Interface (2)

- IDL-Definitionen für das DII



- Mit Hilfe der Methode create_request wird ein Requestobjekt erzeugt. Dabei werden diesem die bereits gesammelten Aufrufparameter übergeben.
- An dem Requestobjekt können dann mit verschiedenen Methoden, die entsprechenden Aufrufe am eigentlichen Objekt ausgeführt werden:

invoke	Aufruf als RPC
send_oneway	Aufruf als Datagramm ohne Antwort
send_deferred	asynchroner Aufruf
get_response	Warten auf Antwort
poll_response	Prüfen, ob Antwort bereits vorliegt

5 Dynamic Skeleton Interface (DSI)

- DSI ermöglicht das Entgegennehmen von Methodenaufrufen für Objekte, deren Schnittstelle nur dynamisch ermittelbar ist, z.B. für
  - ◆ Bridges zu anderen ORBs
    - Bridges stellen eine Verbindung zu anderen ORBs her. Sie haben ein „Bein“ auf jeder Seite und bilden eine „Brücke“.
  - ◆ CORBA-gekapselte Datenbank
  - ◆ dynamisch erzeugte Objekte und Schnittstellen
- Anhand der Parameter wird das Objekt und dessen Signatur ermittelt
  - In der Implementierung wird der Aufruf an eine registrierte Callback-Funktion zugestellt, die dann das entsprechende Objekt aufrufen muß.

★ **Beachte:**
DII und DSI sind jeweils von der Gegenseite nicht von statischen Stubs zu unterscheiden, d.h. voll interoperabel!

6 Object Adaptor

- Lokaler Repräsentant von ORB-Dienstes,
  direkter Ansprechpartner eines CORBA Objekts
  - Generiert Objektreferenzen (für neue Objekte)
  - Bildet Objektreferenzen auf Implementierungen ab
  - Bearbeitet einkommende Methodenaufrufe
  - Authentisierung des Aufrufers (Sicherheitsfunktionalität)
  - Aktiviert und deaktiviert Objekt bzw. dessen Implementierung
    - Ein Objekt ist eventuell zwar für das CORBA System präsent, aber noch nicht aktiv, so dass es direkt aufgerufen werden kann. Vor einem Aufruf wird der Object Adaptor dann das Objekt aktivieren.
  - Registriert Serverklassen im Implementation Repository
- CORBA definiert den Portable Object Adaptor
  - Basis-Funktionalität, muss unterstützt werden
- weiteres Beispiel: OODB Adaptor
  - Anbindung einer objektorientierten Datenbank an CORBA
  - Der OODB Adaptor repräsentiert alle Objekte in der Datenbank als CORBA-Objekte und vermittelt Aufrufe

7 Implementation Repository

- Datenbank, die Implementierungen speichert
  - Daten zur Lokalisierung und Aktivierung der Implementierung
  - Debugging- und Verwaltungsinformationen
  - etc.
- Abfrage der Datenbank zum
  - Implementierung der Methode *get_implementation* eines jeden Objekts
    - Dies entspricht der Methode get_interface beim Interface Repository.
- Schreiben der Datenbank
  - externe Tools
  - Object Adaptor schreibt beim Start eines Servers

- Das Implementation Repository ist implementationsabhängig und damit nicht über mehrere CORBA Implementierungen hinweg kompatibel. Vielfach wird es nicht implementiert.
- Meist nimmt es lediglich die Aktivierungsinformationen für den POA auf.

8 Inter-ORB-Kommunikation

- GIOP – General Inter-ORB Protocol
  - ◆ Basisprotokoll zur Interaktion zweier ORBs
  - ◆ Common Data Representation (CDR) konvertiert IDL-konforme Parameter in einen seriellen Bytestrom
  - ◆ IIOP – Internet Inter-ORB Protocol (GIOP über TCP/IP; Implementierung vorgeschrieben)
  - ◆ andere Implementierungen von GIOP möglich
- ESIOP – Environment Specific Inter-ORB Protocols, z.B. DCE/ESIOP
  - ◆ Inter-ORB Protokoll auf der Basis von DCE RPC
- IOR – Interoperable Object Reference
  Interoperable Repräsentation einer Objektreferenz
  - ◆ String
  - ◆ verschiedene Profiles, u.a. IIOP
    - Eine Objektreferenz besteht eventuell aus verschiedenen Profiles, d.h. aus verschiedenen gleichwertigen Beschreibungen einer Objektreferenz. Diese Beschreibungen sind jedoch meist für verschiedene Protokolle. Eine CORBA-Implementierung kann so beispielsweise ein lokales Profile benutzen, in dem eine proprietäre Protokolladresse für ein Objekt kodiert ist, und gleichzeitig ein IIOP-Profile angeben, das beschreibt, wie das gleiche Objekt über IIOP von außen erreichbar ist.

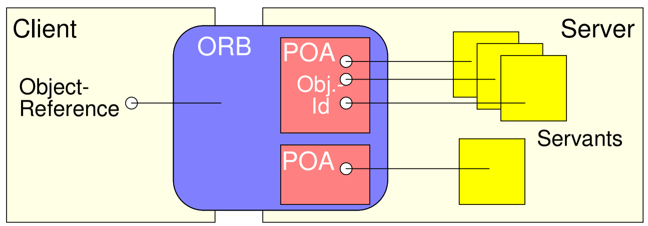
D.11 Portable Object Adaptor (POA)

1 Ziele

- Portabilität von Objektimplementierungen zwischen verschiedenen ORBs
- Trennung von Objekt (CORBA-Objekt mit seiner Identität) und Objektimplementierung
  - ➤ eine Objektimplementierung kann mehrere CORBA-Objekte realisieren
  - ➤ Objektimplementierung kann bei Bedarf dynamisch aktiviert werden
  - ➤ persistente CORBA-Objekte (Objekte, die Laufzeit eines Servers überdauern)
- Mehrere POA-Instanzen (mit unterschiedlichen Strategien) innerhalb eines Servers möglich

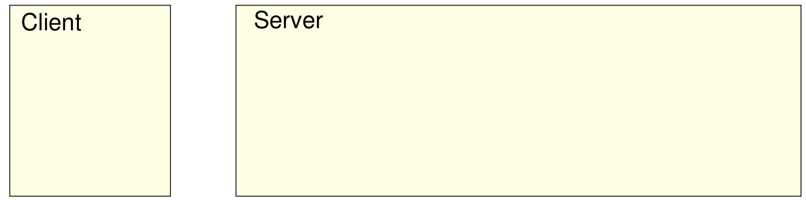
2 Terminologie

- ***Server***: Ausführungsumgebung (Prozess) für CORBA-Objekte
- ***Servant***: konkretes Sprachobjekt, das CORBA-Objekt(e) implementiert
- ***Object***: abstraktes CORBA-Objekt (mit Schnittstelle und Identität)
- ***Object Id***: Identität, mit der ein POA ein bestimmtes *Object* identifiziert
- ***POA***: Verwaltungseinheit innerhalb eines Servers
  - Namensraum für Object-Ids und weitere POAs
  - setzt Aufruf an einem *Object* in einen Aufruf an einem Servant um
- ***Object Reference***: Enthält Informationen zur Identifikation von Server, POA und Object



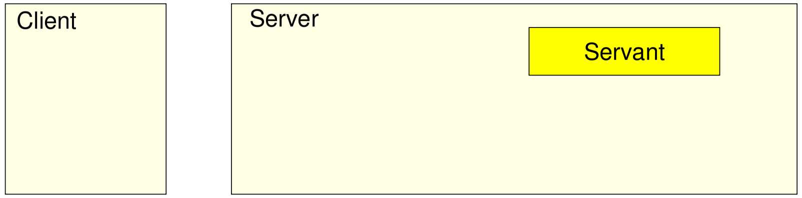
3 Erzeugung eines CORBA-Objekts

- Anwendung erzeugt ein Objekt



3 Erzeugung eines CORBA-Objekts

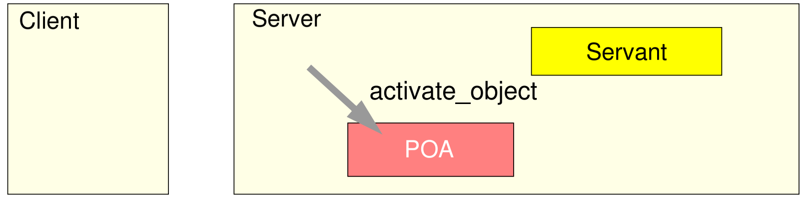
- Anwendung erzeugt ein Objekt



- ◆ 1. Servant wird erzeugt

3 Erzeugung eines CORBA-Objekts

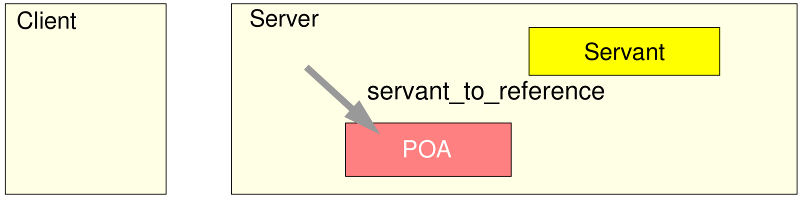
- Anwendung erzeugt ein Objekt



- ◆ 1. Servant wird erzeugt
- ◆ 2. Servant wird mit Object-Id versehen

3 Erzeugung eines CORBA-Objekts

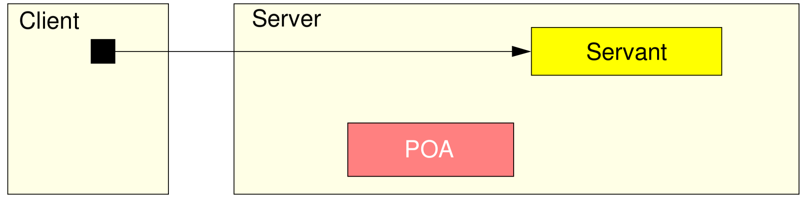
- Anwendung erzeugt ein Objekt



- ◆ 1. Servant wird erzeugt
- ◆ 2. Servant erhält eine Object-Id
- ◆ 3. Servant wird mit Object-Reference versehen (dadurch wird er zum CORBA-Objekt)

3 Erzeugung eines CORBA-Objekts

- Anwendung erzeugt ein Objekt



- ◆ 1. Servant wird erzeugt
- ◆ 2. Servant erhält eine Object-Id
- ◆ 3. Servant wird mit Object-Reference versehen (dadurch wird er zum CORBA-Objekt)
- ◆ 4. Reference kann nun über Name Service verfügbar gemacht werden oder als Ergebnis eines Aufrufs an einen Client übergeben werden. Dort wird dann ein Client-Stub zur Weiterleitung der Aufrufe an den Server erzeugt.

➥ eine Object-Reference beim Client steht für ein bestimmtes CORBA-Objekt — nicht unbedingt für einen bestimmten Servant!

4 Alternativen zur Aktivierung von CORBA-Objekten

- Implizite Aktivierung
  - POA erzeugt automatisch eine Object -Reference wenn ein (nicht aktivierter) Servant als Parameter oder Ergebnis "nach aussen" übergeben wird
- Aktivierung "on demand"
  - Object-Reference wird erzeugt ohne dass ein Servant existiert
    - damit entsteht ein abstraktes CORBA-Objekt, zunächst ohne Implementierung
  - Verschiedene Möglichkeiten, ankommende Aufruf zu bearbeiten:
    - Ein registrierter "Default Servant" bearbeitet den Aufruf (z. B. wenn Datenbank-Einträge als CORBA-Objekte exportiert werden)
    - Ein "Servant Manager" (vom Benutzer zu implementieren) liefert einen Servant (der Servant Manager kann den Servant ggf. dynamisch erzeugen)

5 Deaktivierung und Aktivierung

- Objekte können eine POA-Instanz überleben: Persitente Objekte
  - Servant kann dekativiert werden
  - POA kann deaktiviert werden
  - Server kann gestoppt werden
- POA kooperiert mit "Location Forwarding Service" und dem Implementation Repository
  - Bei der Deaktivierung werden die für die Aktivierung notwendigen Informationen im Implementation Repository aufgehoben
  - Aufrufe an deaktivierte Objekte gehen zunächst an den Forwarding Service. Dieser startet einen neuen Server.
  - Der Server erhält den weitergeleiteten Aufruf und aktiviert einen POA.
  - Der POA aktiviert einen Servant und übergibt ihm den Aufruf.

6 POA-Strategien (Policies)

- Eine POA-Strategie ist jeweils einer POA-Instanz zugeordnet
  - Innerhalb eines Servers kann es mehrere POA-Instanzen, jeweils mit einer anderen Strategie geben.
- Aktivierungs-Methode ist in solch einer Strategie festgelegt
  - Servant retention policy
  - Request processing policy
  - Implicit activation policy
- Weitere Strategien:
  - Thread policy: multi-threaded oder single-threaded
  - ID uniqueness policy: ob eindeutige IDs benutzt werden
  - ID assignment policy: IDs werden durch Benutzer bzw. System erzeugt
  - Object lifespan policy: transient bzw. persistent

D.12 CORBA Services

- Basisdienste eines verteilten Systems – Erweiterungen des ORB

Collection Service	**Persistent Object Service**
Concurrency Service	**Property Service**
Enhanced View of Time	**Query Service**
Event Service	**Relationship Service**
Externalization Service	**Security Service**
Naming Service	**Time Service**
Licensing Service	**Trading Object Service**
Life Cycle Service	**Transaction Service**
Notification Service	

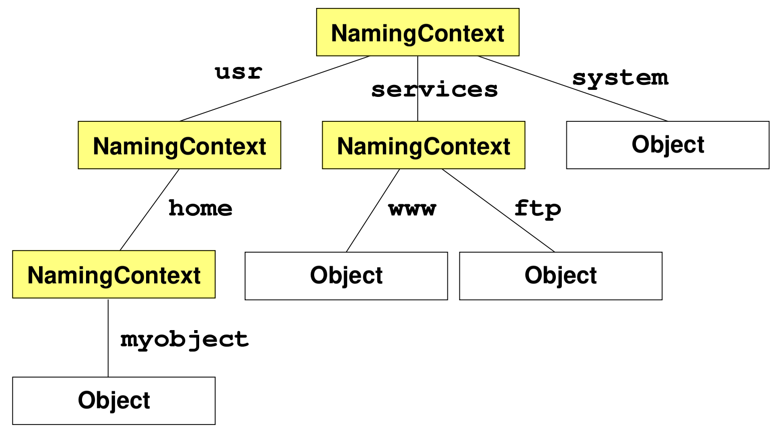
- Services sind über IDL-Schnittstellen aufrufbar
  - Keine zusätzlichen Konstrukte erforderlich; Methodenaufruf reicht aus

1 Naming Service

- CORBA definiert einen hierarchischen Namensdienst ähnlich dem UNIX Dateinamensdienst
  - Namensraum ist baumförmig
  - Name besteht aus mehreren Komponenten (Silben - syllables)
    z. B. < "usr"; "home"; "myobject" >
    (UNIX: "/usr/home/myobject")
  - Schnittstellen festgelegt durch IDL-Beschreibungen

1 Namensdienst (2)

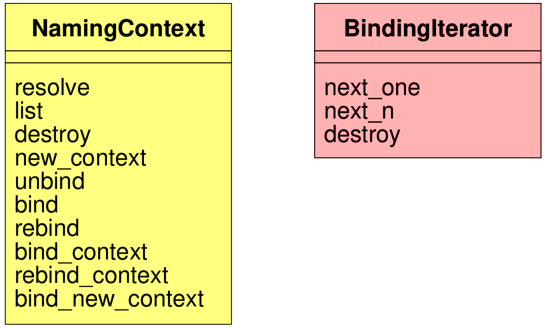
- Beispielbaum



- In diesem Baum gibt es beispielsweise den Namen
  < "usr" ; "home" ; "myobject" >
- Dieser Name besteht aus drei Komponenten. In unserer Notation wurden sie mit Strichpunkten abgetrennt. Es ist zu beachten, dass CORBA keine solche Notation definiert. Es ist lediglich festgelegt, dass der Name aus Komponenten besteht. Eine andere Notation könnte sein
  /usr/home/myobject
  wie aus UNIX bekannt.
- Bei der Auflösung dieses Namens erlangt man die Objektreferenz auf das entsprechende Objekt.
- Die Referenz auf den root-NamingContext bekommt man über eine Abfrage beim ORB
  `orb->resolve("NamingService");`

1 Namensdienst (3)

- Kontext- und Iteratorschnittstelle



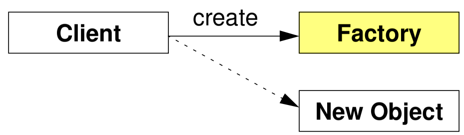
- Die Klasse NamingContext stellt so etwas wie ein Verzeichnis oder Directory dar. Man kann Namenskomponenten definieren und diese entweder mit einem Objekt oder einem anderen Kontext verbinden. Im Falle eines Kontexts entsteht der besagte Baum.
- Beim Auflisten eines Kontexts kann eine maximale Anzahl von Namenseinträgen angegeben werden, die der Aufrufer haben möchte. Überbleibende Einträge werden dann in einen BindingIterator gepackt, der zusätzlich zurückgegeben wird. Über diesen kann man dann auf die folgenden Einträge zugreifen.

2 Life Cycle Service

- Lebenszyklus eines Objekts
  - Erzeugung
  - Kopieren, Verlagern
  - Löschen
- Life Cycle Service definiert eine gemeinsame Schnittstelle für Operationen während des Lebenszyklus

★ Modell des Lebenszyklus

- Erzeugung eines Objekts:

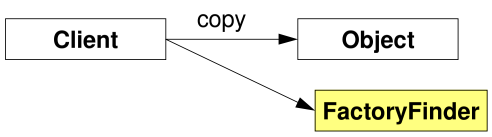


Aufruf einer create-Methode an einem Factory-Objekt

- Die Factory wird aufgerufen, erzeugt ein neues Objekt und gibt dessen Objektreferenz an den Aufrufer zurück.
- Das Protokoll bzw. das Interface der Factory ist in der Regel abhängig vom zu erzeugenden Objekt und nicht festgelegt.

2 Life Cycle Service (2)

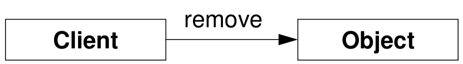
◆ Kopieren und Verlagern:



FactoryFinder bestimmt den Ort

- Client kennt einen FactoryFinder
- Der FactoryFinder bestimmt den Ort und oder die Region, in der die Kopie erzeugt wird bzw. in die das Objekt verlagert werden soll.
- Orte könnten sein: ein bestimmter Rechner, eine bestimmte Rechnergruppe etc.

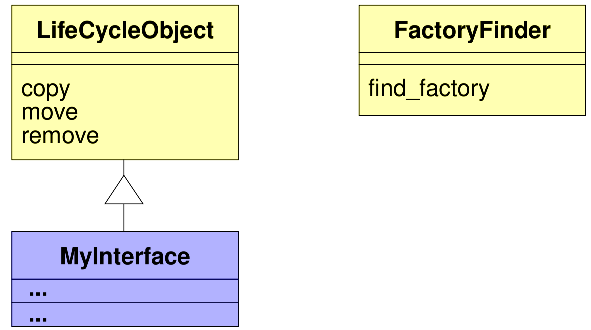
◆ Löschen:



- Mit dem Aufruf von remove löscht sich das Objekt.

2 Life Cycle Service (3)

■ Objekte müssen das Interface LifeCycleObject implementieren



◆ **`copy`** und **`move`** benötigen ein **`FactoryFinder`**-Objekt um ein **`Factory`**-Objekt zu finden, das dann die Kopie bzw. das verlagerte Objekt erzeugt

◆ **`remove`** löscht ein Objekt

2 Life Cycle Service (4)

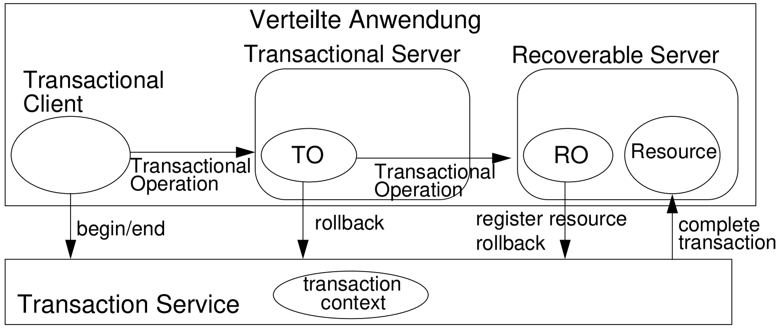
- Implementierung am Beispiel copy
  - ◆ Client ruft copy-Methode auf und übergibt einen FactoryFinder
  - ◆ Die copy-Methode stellt entweder der Programmierer des Objekts oder die CORBA-Implementierung bereit
  - ◆ Die copy-Methode ruft den FactoryFinder auf, sucht eine passende Factory aus und erzeugt mit ihrer Hilfe ein neues Objekt.
  - ◆ Das neue Objekt wird mit den Daten des bestehenden Objekts initialisiert.
- ▲ Nach aussen klare Schnittstelle
- ▲ Nach innen offen und implementationsabhängig,
  z. B. Protokoll zwischen copy-Methode und Factory
- In CORBA 2.0 Berücksichtigung von Teil-Ganze-Beziehungen und ähnlichen
  (Deep Copy und Shallow Copy)
  - Die Beziehungen zwischen Objekten können registriert werden. Bei einer Kopier- oder Löscheoperation wird dann ein ganzer Objektgraph kopiert oder gelöscht.

3 Object Transaction Service (OTS)

- Transaktionen: "ACID":
  - atomic (alles oder nichts)
  - consistent (Neuer Zustand erfüllt Konsistenzbedingungen)
  - isolated (Isolierung: keine Zwischenzustände sichtbar)
  - durable (Persistenz)
- Eine Transaktion => mehrere Objekte, mehrere Requests:
  Bindung an einen *"Transaction context"*
  - ◆ wird normalerweise *implizit* an alle Objekte weitergereicht
  - ◆ auch explizite Weitergabe durch Client möglich (Spez. in IDL)
- Typischer Ablauf:
  - ◆ Beginn der Transaktion erzeugt *Transaction context* (an den Client-Thread gebunden)
  - ◆ Ausführung von Methoden (implizit an die Transaktion gebunden)
  - ◆ Schliesslich: Client beendet die Transaktion (commit/roll back)

3 Object Transaction Service (OTS) (2)

- Bestandteile einer Anwendung, die vom OTS unterstützt wird:
  - Transactional Client
  - Transactional Objects (TO)
  - Recoverable Objects (RO)
  - Transactional Servers
  - Recoverable Servers



3 Object Transaction Service (OTS) (3)

- Zugriff auf Transaction Service über **Current**-Objekt
  - **orb.resolve_initial_reference("TransactionCurrent");**

```
interface Current : CORBA::Current {
   void begin() raises(SubtransactionsUnavailable);
   void commit(in boolean report_heuristics)
      raises(NoTransaction,HeuristicMixed,HeuristicHazard);
   void rollback() raises(NoTransaction);
   void rollback_only() raises(NoTransaction);

   Status get_status();
   string get_transaction_name();
   void set_timeout(in unsigned long seconds);

   Control get_control();
   Control suspend();
   void resume(in Control which) raises(InvalidControl);
};
```

3 Object Transaction Service (OTS) (4)

- *Transaction Context*: `Control`-Objekt

```
interface TransactionFactory {
   Control create(in unsigned long time_out);
   Control recreate(in PropagationContext ctx);
};

interface Control {
   Terminator get_terminator() raises(Unavailable);
   Coordinator get_coordinator() raises(Unavailable);
};

interface Terminator {
   void commit(in boolean report_heuristics) raises(...);
   void rollback();
};
```


4 CORBA Services - Zusammenfassung

- Von OMG standardisierte Spezifikationen von Dienste für Probleme, die in verteilten Systemen häufig auftreten
- Spezifikationen legen fest:
  - Immer: Einheitliche Schnittstelle (IDL)
  - Meistens: generelle Vorgehensweise bei der Problemlösung
  - Manchmal: konkrete Strategien (oft auch offen gelassen bzw. implementierungsabhängig)
- Weitere Dokumentation/Informationen:
  - http://www.omg.org/technology/documents/formal/corbaservices.htm
  - http://developer.java.sun.com/developer/onlineTraining/corba/corba.html